ALTO: A Personal Computer System Hardware Manual

May, 1979

Abstract

This manual is a revision of the original description of the Alto: "Alto, A Personal Computer System." It includes a complete description of the Alto I and Alto II hardware and of the standard microcode (I:24, II:3).

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Alto Hardware Manual

Table of Contents

1.0 INTRODUCTION

This document is a description of the Alto, a small personal computing system originally designed at PARC. By "personal computer" we mean a non-shared system containing sufficient processing power, storage, and input-output capability to satisfy the computational needs of a single user.

A basic Alto system is:

* An 875-line television monitor, with a viewing area of about 8½" x 11", oriented with the long tube dimension vertical. The controller provides a 606 by 808 point display which is refreshed from main memory at 60 fields (30 frames) per second. It has programmable polarity, a low resolution mode which conserves memory space, and a 16 by 16 cursor whose position and content are under program control.
* An unencoded 64-key keyboard.
* A mouse (pointing device) and five-finger keyset.
* Up to two Diablo Model 31 disk drives or a Model 44 disk drive.
* An interface to the Ethernet, a 3 Mbps local network that can connect up to 256 Altos and other computers separated by as much as a mile. Most Ethernets are interconnected by gateways and leased lines to form a nationwide internet.
* A microprogrammed processor which controls the disk, display and Ethernet, and emulates an instruction set. The standard instruction set for which emulation microcode is supplied in the microinstruction ROM is described in section 3.0.
* 64K 16 bit words of 850ns error corrected semiconductor memory, expandable to 256K.
* 1K microinstruction RAM that can be read and written with special microcode to extend the standard instruction set or to emulate a different instruction set or to drive special I/O devices.
* The processor, disk, and their power supplies are packaged in a small cabinet. The other I/O devices may be a few feet away, and are pleasingly packaged for desk top use.

Some options:

* An expanded microinstruction memory consisting of either 2K of PROM or 3K of RAM.
* A Diablo HyType printer.
* A Versatec Printer/Plotter.
* A controller for CalComp Trident disk drives.
* A controller for MDS and Kennedy tape drives.
* An Orbit, the controller for a vast array of laser-scanned printers.
* Communications controllers for BBN-1822, SDLC, BiSync and Async.

The remaining sections of this document will discuss the hardware and microcode of the standard configuration Alto. At present, two slightly different versions of the Alto exist: the Alto I and the Alto II. Most passages of this document pertain to both machines; those that apply to one only are clearly marked.

This document does not deal with the numerous non-standard peripheral devices that have been interfaced to the Alto. Non-standard interfaces and their designers are tabulated in an appendix.

1.1 Guide to this Document

This document is a comprehensive description of the Alto. Information about hardware, microcode, and CPU programming is sprinkled throughout. Programmers interested primarily in the CPU emulator should concentrate on the sections labeled with an asterisk in the table of contents.

1.2 People

The Alto was originally designed by Charles P. Thacker and Edward M. McCreight and was based on requirements and ideas contributed by Alan Kay, Butler Lampson and other members of PARC's Computer Sciences Laboratory and Systems Sciences Laboratory. Bob Metcalfe and David Boggs designed the Ethernet; Severo Ornstein and Bob Sproull designed the Orbit; Roger Bates designed the Trident controller; David Boggs designed the tape controller; Tat Lam, Dick Lyon, Ed McCreight and Dan Swinehart designed the Audio Board; Larry Stewart designed the BBN-1822 interface.

The machine was re-engineered as the Alto II for ITG/SDD to a specification developed by John Ellenby. The engineering and production were carried out by EOD Special Programs Group, managed by Doug Stewart and coordinated on behalf of PARC and SDD by John Ellenby. The members of EOD/SPG who worked on the project are Doug Stewart, Ron Cude, Ron Freeman, Jim Leung, Tom Logan, Bob Nishimura, Abbey Silverstone, Nathan Tobol, and Ed Wakida.

This hardware manual has had a long history of modification and extension and has benefited from endless toil by numerous individuals. The original manual was written by Chuck Thacker and Ed McCreight. The last major revision was edited by Bob Sproull and Diana Merry. The present document is the responsibility of Ed McCreight, David Boggs, and Ed Taft.

1.3 Conventions and Notation

Numbers in this document are decimal unless followed by "B"; thus 10 = 12B.

Bits in registers are numbered from the most significant bit (0) toward the least significant bit. Fields within registers are given by following the register name with a pair of numbers in brackets: IR[a-b] describes the b-a+1 bit field of the IR register beginning with bit a and ending with bit b inclusive. IR[a] is short for IR[a-a].

The symbol "←" is used to mean "is replaced by." Thus IR[4-5] ← 2 means that the 2-bit field of IR including bits 4 and 5 is replaced by the bit values 1 and 0 respectively. The symbol "=" is used as an equality test.

Memory is by convention divided into 256-word "pages." Page n thus contains addresses 256*n to 256*n+255 inclusive. The notation "rv(adr)" is used, as in BCPL, to denote "the contents of the memory location with address adr."

2.0 MICROPROCESSOR

This section describes the Alto microprocessor structure. If your programming needs on the Alto do not extend to writing new microcode, this section is best left untackled. If you do need to decipher what follows, it may be helpful to have a listing of the "standard" Alto microcode at your side.

The microprocessor is shown schematically in Figures 1 and 2. A principal design goal in this system was to achieve the simplest structure adequate for the required tasks. As a result, the central portion of the processor contains very little application-specific logic, and no specialized data paths. The entire system is synchronous, with a clock interval of approximately 170 nsec. All microinstructions require one cycle for their execution.

A second design goal was to minimize the amount of hardware in the I/O controllers. This is achieved by doing most of the processing associated with I/O transfers with microprograms. To allow devices to proceed in parallel with each other and with CPU activity, a control structure was devised which allows the microprocessor to be shared among up to 16 fixed priority tasks. Switching among tasks requires very little overhead, and occurs typically every few microseconds.

2.1 Arithmetic Section

The arithmetic section of the processor consists of two 32-word by 16-bit register files R and S, and five registers, T, L, M, MAR, and IR. The registers are connected to the memory and to an ALU with a 16-bit parallel bus. For historical reasons, the S and M registers are viewed as part of the microinstruction RAM and are described in section 8.

The ALU is a SN74181 type, restricted so that it can do only 16 arithmetic and logical functions. The ALU output feeds the L, M, and MAR registers. T may also be loaded from the ALU output under certain conditions. L is connected to a shifter capable of left and right shifts by one place, and cycles of 8. It has a mode in which it does the peculiar 17-bit shifts of the standard instruction set, and a mode which allows double-length shifts to be done.

The IR register is used by the emulator to hold the current emulated instruction -- see section 3.5.

Attached to the bus is a 256-word read only memory (ROM) which holds arbitrary 16-bit constants.

The fields of the 32-bit microinstruction are:

FIELD	NAME	MEANING
0-4	RSELECT	R Register Select
5-8	ALUF	ALU Function
9-11	BS	Bus Data Source
12-15	F1	Function 1
16-19	F2	Function 2
20	T	Load T
21	L	Load L & M
22-31	NEXT	Next microinstruction address (subject to modifiers)

When microprogramming the Alto, it is important to understand where the machine's state resides and how it changes. At the beginning of a microinstruction cycle, the various registers (principally T, L, M, and IR, but also various bits of state such as ALUC0) contain values that remain unchanged throughout execution of the microinstruction. During this time, the various non-state-retaining data paths and elements, such as the bus, ALU, and shifter, compute results based entirely on the initial values of these

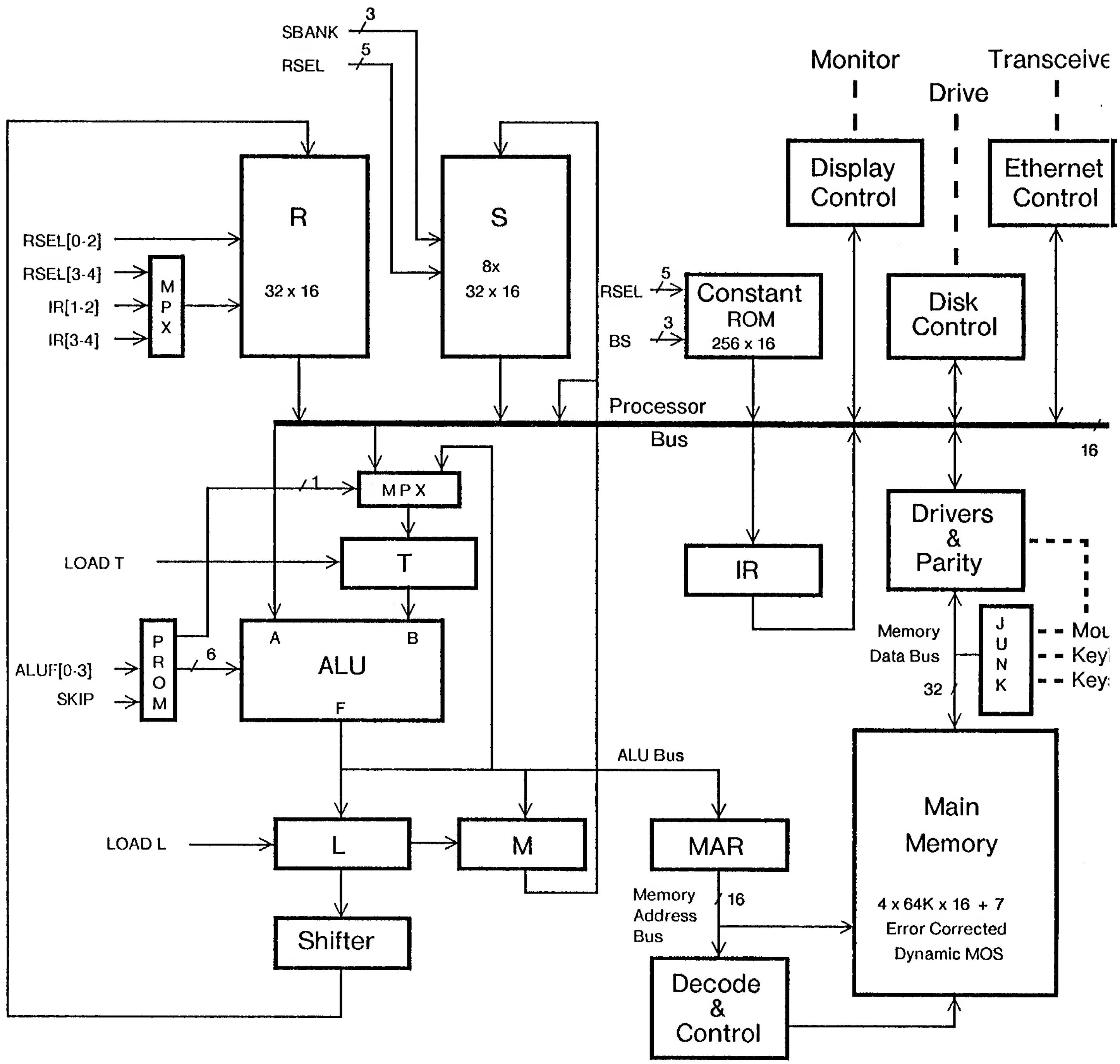


Figure 1 -- Processor Data Paths

RSEL	ALUF	BS	F1	F2	T	L	NEXT

registers. However, the registers themselves do not change.

At the end of the cycle, if the microinstruction specifies that one or more registers be loaded, they are loaded instantaneously and simultaneously with the newly-computed values. These then serve as the initial register values for the next microinstruction. As a result, it is possible (and in fact very common) to both read and load a register during the same microinstruction. The R registers behave similarly except that it is not possible to both read and load an R register during the same microinstruction.

R SELECT

The R select field specifies one of the 32 R cells to be loaded or read under control of the bus source field, or, in conjunction with the bus source field, one of the 256 locations to be read from the constant ROM. The R field is also used to address registers in S -- see section 8.

The low order two bits of the R address (but not the constant ROM address) may be taken from fields in IR under control of the functions. This allows the emulator to address its central registers easily.

ALU FUNCTIONS

The ALUF field controls the SN74181 ALU. This device can do a total of 48 arithmetic and logical operations, most of which are relatively useless. The 4-bit field is mapped by a PROM into the 16 most useful functions.

ALUF	T	FUNCTION	S3	S2	S1	S0	M	C	OPERATION
0	*	BUS	1	1	1	1	1	0	A
1		T	1	0	1	0	1	0	B
2	*	BUS OR T	1	1	1	0	1	0	A+B
3		BUS AND T	1	0	1	1	1	0	AB
4		BUS XOR T	0	1	1	0	1	0	A XOR B
5	*	BUS + 1	0	0	0	0	0	0	A PLUS 1
6	*	BUS - 1	1	1	1	1	0	1	A MINUS 1
7		BUS + T	1	0	0	1	0	1	A PLUS B
10B		BUS - T	0	1	1	0	0	0	A MINUS B
11B		BUS - T - 1	0	1	1	0	0	1	A MINUS B MINUS 1
12B	*	BUS + T + 1	1	0	0	1	0	0	A PLUS B PLUS 1
13B	*	BUS + SKIP	0	0	0	0	0	SKIP	A PLUS 1
14B	*	BUS . T (AND)	1	0	1	1	1	0	AB
15B		BUS AND NOT T	0	1	1	1	1	0	A & NOT B
16B-17B		UNDEFINED							

If T is loaded in an instruction containing an ALUF with a * in the T column, it will be loaded from the ALU output rather than from BUS.

S3-S0 selects the function; M selects logical or arithmetic mode by controlling carry propagation; C is the carry into the LSB. The carry output is forced to zero during logical operations (M=0). BUS is the A input to the ALU; T is the B input.

BUS SOURCES

The bus data source (BS) field specifies one of 8 data sources for the bus:

BS	NAME	SOURCE
0	←RName	Read R
1	RName←	Load R from shifter output (see below)
2	(None)	Enables no source to the BUS, leaving it all ones
3	Task-specific	Performs different functions in different tasks.

4	Task-specific	Performs different functions in different tasks.
5	←MD	Memory data
6	←MOUSE	BUS[12-15]← MOUSE; BUS[0-13]← -1
7	←DISP	IR[8-15], possibly sign extended (see section 3.5)

RName← is not logically a source, but because it is gated to the bus during both reading and writing, it is included in the source specifiers. Loading R forces the BUS to 0 so that an ALU function of 0 and T may be executed simultaneously.

The bus has the property that if more than one source is gated to it during a single microinstruction, it computes the AND of the source values. This is true regardless of the means by which the sources are enabled (BS, F1, or F2).

This bus source decoding is not performed if F1=7 or F2=7. These functions use the BS field to provide part of the address to the constant ROM.

SPECIAL FUNCTIONS

The two function fields specify the address modifiers, register load signals (other than those for R, S, L, M and T), and other special conditions required in the processor. The first eight conditions specified by each field (except BLOCK) are interpreted identically by all tasks, but the interpretation of the second eight depends on the active task. The task-independent functions are given below; the task-specific functions are included with the task descriptions.

FUNCTION 1:

F1	NAME	MEANING
0	---	No Activity
1	MAR←	Load MAR from ALU output; start main memory reference (see section 2.3).
2	TASK	Switch tasks if higher priority wakeup is pending (see section 2.4).
3	BLOCK	Disable the current task until re-enabled by a hardware-generated condition. Note: this function is reserved by convention only; it is *not* done by the microprocessor.
4	←L LSH 1	SHIFTER OUTPUT will be L shifted left one place*
5	←L RSH 1	SHIFTER OUTPUT will be L shifted right one place*
6	←L LCY 8	SHIFTER OUTPUT will be L rotated left 8 places*
7	←CONSTANT	Put on the bus the constant from the constant ROM location addressed by RSELECT.BS

*Modified by DNS (Do Novel Shifts) function, and MAGIC function. L LSH 1 and L RSH 1 ordinarily shift a zero into the vacated bit position.

FUNCTION 2:

F2	NAME	MEANING
0	---	No Activity
1	BUS=0	NEXT←NEXT OR (if (BUS=0) then 1 else 0).
2	SH<0	NEXT←NEXT OR (if (SHIFTER OUTPUT<0) then 1 else 0).*
3	SH=0	NEXT←NEXT OR (if (SHIFTER OUTPUT=0) then 1 else 0).*
4	BUS	NEXT←NEXT OR BUS[6-15]

5	ALUCY	NEXT←NEXT OR ALUC0. ALUC0 is the carry produced by the ALU during the most recent microinstruction that loaded L. It is *not* the carry produced during execution of the microinstruction that contains the ALUCY function.
6	MD←	Deliver BUS data to memory (see section 2.3)
7	←CONSTANT	Same as F1=7

*Note that the value of the SHIFTER OUTPUT is determined by the value of L as the microinstruction *begins* execution and the shifter function (L LSH 1, L RSH 1, or L LCY 8) specified during the *current* microinstruction (if no shifter function is specified, the shifter output is equal to L).

2.2 Constant Memory

The constant memory is a 256 x 16 PROM that holds arbitrary constants. The constant memory is gated to the bus by F1=7, F2=7, or BS≥4. The constant memory is addressed by the (8 bit) concatenation of RSELECT and BS. The intent in enabling constants with BS≥4 is to provide a masking facility, particularly for the ←MOUSE and ←DISP bus sources. This works because the processor bus ANDs if more than one source is gated to it. Up to 32 such mask constants can be provided for each of the four bus sources ≥4.

Alto I: Note that it is not possible to use a constant other than -1 with the ←MD bus source, because memory parity is calculated on the bus, and a parity error will result if bits are masked off in a word fetched from memory.

2.3 Main Memory

Main memory references are handled differently on Alto I and Alto II. It is, however, possible to write most microcode so that it will operate correctly on both machines.

BASICS

Memory is addressed by a 16-bit number that refers to a 16-bit word in the memory. Addresses 0 through 176777B are true memory storage locations; addresses 177000B through 177777B are used to control I/O devices that are attached to the Alto memory bus. Some operations on memory are performed on "double-words." The double-word beginning at location adr (adr is even) is a 32-bit quantity equivalent to the 16-bit contents of location adr, together with the 16-bit contents of location adr+1. (Double-word references operate correctly only on true memory locations, not on I/O device locations.)

MEMORY REFERENCES

Alto I and Alto II: A memory reference is initiated by executing F1=1, MAR←. The results of a read operation are delivered somewhat later onto the bus with BS=5, ←MD. A store into the addressed memory location is achieved with F2=6, MD←. The microprogram partially controls memory timing, and must observe certain rules to insure correct operation.

a) A minimum of one microinstruction must intervene between the initiation of a memory reference and an MD← or ←MD.

b) On both Alto I and Alto II, memory cycles last a total of 5 micro-cycles, although double-word operations may extend the memory cycle to take a total of 6 micro-cycles. Although the exact details of memory timing differ on Alto I and Alto II, both machines share the property that the processor will suspend execution of microinstructions if the memory interface cannot process the function (MAR←, MD← or ←MD) specified; processing will resume as soon as the interface is free. It is permissible to "abandon" a memory reference that has already been started simply by not referencing MD within the first 5 cycles, or by starting a new memory reference with MAR←.

c) The memory checks parity on all fetches, unless the cycle is a refresh cycle or the address is between 177000B and 177777B inclusive, in which case an I/O device is being referenced. Parity errors result in activation of a high-priority task (task number 15B) whose purpose is to deal with the error (see section 5.5). The Alto II checks memory parity on store as well as fetch cycles.

d) If RSELECT = 37B during the instruction which starts the memory, a refresh cycle is assumed and all memory cards are activated. This is used by the refresh task.

e) MAR← cannot be invoked in the same instruction as ←MD of a previous access.

In the discussion that follows, we assume that a memory reference has been started with MAR←, and we designate this instruction (micro)cycle 1. Examples of proper sequences are given below.

Alto I

f) During cycle 5, if F2=6, MD←, a store of bus data into the word addressed by MAR will occur. The MD← may not be issued later than cycle 5. (Note: Some Alto I's have been modified to allow a "double-word store." On these machines, it is permissible to issue two MD← instructions in a row, the first coming in cycle 5, and the second in cycle 6. If MAR is loaded with an even address adr, the two words will be stored at adr and adr+1 respectively.)

g) During cycle 5 of a reference, if BS=5, ←MD, the reference is a fetch of the word addressed by MAR. During cycle 6, if BS=5, ←MD, the odd word of the doubleword addressed by MAR is delivered. If MD is referenced during cycle 6, it also must have been referenced (by either ←MD or MD←) during cycle 5.

Alto II

f) During cycle 4, if F2=6, MD←, a store of bus data into the word addressed by MAR will occur. The MD← may not be issued later than cycle 4. Alto II's allow a "double-word store:" it is permissible to issue two MD← instructions in a row, the first coming in cycle 3, and the second in cycle 4. If MAR is loaded with an address adr, the two words will be stored at adr and (adr XOR 1) respectively.

g) During cycle 5, if BS=5, ←MD, the reference is a fetch of the word addressed by MAR. During cycle 6, if BS=5, ←MD, the other word of the doubleword addressed by MAR is delivered. Again, if MAR is loaded with address adr, the two words fetched will be from location adr and (adr XOR 1) respectively.

h) Because the Alto II latches memory contents, it is possible to execute ←MD anytime after cycle 5 of a reference and obtain the results of the read operation.

EXAMPLES

Because the description above is a bit terse, we shall give several examples for Alto I operation, for Alto II operation, and for coding schemes that will work properly on both kinds of Altos. In the coding examples, REQUIRED stands for some microinstruction (you supply it) that must appear in the sequence; SUSPEND stands for a microinstruction which if omitted will cause execution to suspend for one cycle because the memory interface is not ready; OPTIONAL stands for a microinstruction which may be omitted without penalty. The notation ANY will be used to stand for an arbitrary 16-bit address; EVEN will stand for an even 16-bit address. All of these examples apply to extended memory references also (described in the next section); simply substitute XMAR for MAR.

Simple fetch:

```
Alto I              Alto II

MAR←ANY;            MAR←ANY;
REQUIRED;           REQUIRED;
SUSPEND;            SUSPEND;
SUSPEND;            SUSPEND;
whereever←MD;       whereever←MD;
```

Simple store:

```
Alto I              Alto II

MAR←ANY;            MAR←ANY;
REQUIRED;           REQUIRED;
SUSPEND;            OPTIONAL;
SUSPEND;            MD←whatever;
MD←whatever;
```

Simple store, followed immediately by another memory cycle:

```
Alto I              Alto II             Alto II

MAR←ANY;            MAR←ANY;            MAR←ANY;
REQUIRED;           REQUIRED;           REQUIRED;
SUSPEND;            REQUIRED;           MD←whatever;
SUSPEND;            MD←whatever;        SUSPEND;
MD←whatever;        SUSPEND;            SUSPEND;
MAR←ANY;            MAR←ANY;            MAR←ANY;
...                 ...                 ...
```

Double-word fetch:

```
Alto I              Alto II

MAR←EVEN;           MAR←ANY;
REQUIRED;           REQUIRED;
SUSPEND;            SUSPEND;
SUSPEND;            SUSPEND;
whereever←MD;       whereever←MD;
whereever←MD;       whereever←MD;
```

Double-word store/fetch:

```
Alto I              Alto II

MAR←EVEN;           MAR←ANY;
REQUIRED;           REQUIRED;
SUSPEND;            SUSPEND;
SUSPEND;            MD←whatever;
MD←whatever;        whereever←MD;
whereever←MD;
```

Double-word store (only on modified Alto Is):

```
Alto I              Alto II

MAR←EVEN;           MAR←ANY;
REQUIRED;           REQUIRED;
SUSPEND;            MD←whatever;
```

```
        SUSPEND;                MD←whatever;
        MD←whatever;
        MD←whatever;
```

The Alto II memory buffering permits a double-word "exchange":

```
        MAR←ANY;
        REQUIRED;
        MD←newContents1;        address = adr
        MD←newContents2;        address = adr XOR 1
        L←MD;                   address = adr
        T←MD;                   address = adr XOR 1
        oldContents1←L, L←T;
        oldContents2←L;
```

Microcode which uses the memory timings below will work on either vintage of Alto:

Simple fetch: (as Alto I).

Simple store: (as Alto II). <<<<< **Nota Bene**

Double-word fetch: (as Alto I).

Double-word store/fetch: (as Alto II).

Others are not possible.

EXTENDED MEMORY

Main memory on Alto IIs can be optionally expanded to up to 256K words in 64K *banks*. Each task has associated with it four extra *bank bits* which are presented to the memory along with the 16 bit addresses generated by the task's microcode. *Normal memory references* are microcoded in the usual way and use two of the bank bits to specify the task's *normal bank*. *Extended memory references* are microcoded slightly differently and use the two other bank bits to specify the task's *alternate bank*. Thus a task can reference 64K very easily, another 64K with a little difficulty, and the other two 64K banks only after loading its bank registers appropriately.

To signal that a memory reference should go to the alternate bank, the microinstruction which loads MAR must also contain F2=6 (MD←). The microassembler will generate this conbination of functions for a clause whose left hand side is XMAR (i.e., XMAR← address will generate an instruction with F1=1 and F2=6).

The bank registers appear as 16 words in the I/O area which can be read and written. Location (177740B + N) is the bank register location for task N. Booting the Alto clears the registers to zeros making all references for all tasks go to bank zero, thus making the machine operate as a standard Alto without the extended memory option. Within a bank register, the layout is as follows:

BR[0-11]	undefined
BR[12-13]	normal reference bank number
BR[14-15]	extended reference bank number

The highest 512 locations in each bank are not mapped by the bank registers and always refer to the I/O area. That means that location 177740B is the emulator's bank register regardless of what the referencing task's bank register contains and regardless of whether it is referenced with a normal or an extended memory reference.

No changes are necessary in order to run the display, disk, or Ethernet in different banks. The easiest and least confusing way to do this is to load the bank registers for all concerned tasks (e.g. DVT, DHT and DWT for the display, or KSEC and KWD for the disk) with some other bank number. Then the device is controlled by the relevant words of page 1 in its bank.

Programs which use the extended memory must first initialize it to have correct parity. This involves disabling parity interrupts, storing something in every word, flushing any parity interrupts that result, and then reenabling parity interrupts. The standard bootstrap loaders initialize bank zero only.

All Alto IIs manufactured starting with the 7th build have the extended memory option but are normally shipped with memory chips for bank zero only. Some earlier Alto IIs have been modified in the field. Machines with the extended memory option have engineering number 3 -- see the description of the VERS instruction.

2.4 Microprocessor Control

Control of the Alto microprocessor is shared among 16 "tasks" arranged in a priority order. The tasks are numbered 0 to 15: 0 is the lowest priority task and 15 is the highest. The lowest priority task is the emulator task which fetches instructions and executes them.

The only state saved for each task is a "micro program counter," MPC. The current task number, saved in the current task register, addresses a 16 by 12 MPC RAM. The result is an MPC for the current task; it is used to address a 1K by 32-bit read-only microinstruction memory (MI ROM0) or a 1K by 32-bit writeable microinstruction memory (MI RAM0), described in section 8. An optional feature of Alto IIs extends the MI ROM to 2K or the RAM to 3K -- see section 8.

BRANCHING

The microprocessor offers a limited branching capability which, although somewhat cumbersome, has proven adequate for chores undertaken by Alto microcode. The basic idea is that special microprocessor functions may modify the NEXT field, and consequently alter the flow of control. Modification is accomplished by ORing various bits into the NEXT field.

Address modification is complicated slightly because the Alto pre-fetches one microinstruction ahead. Consequently, *a branch condition modifies the NEXT field of the microinstruction following the one in which the condition test is placed.* This property is best illustrated with an example:

MI location	MI
100B	F2=2 (SH<0), NEXT=101B
101B	..., NEXT=102B
102B	...
103B	...

When the instruction at location 100B is being executed, the instruction at location 101B has already been fetched. Therefore, the SH<0 test modifies the NEXT field of the on-deck instruction, the one at 101B. Thus the two possible execution sequences are: (1) if $L \geq 0$ on entering the code above: 100B, 101B, 102B; (2) if $L<0$ on entering the code: 100B, 101B, 103B.

TASK SWITCHING

Only one of the 16 tasks is executing microinstructions at any one time. Once a task begins execution, it continues to execute until it invokes a task switch function that enables switching to another task. A task is considered eligible for execution if its hardware-generated "wakeup signal" is asserted (these signals are not accessible to the microprogram). The wakeup signals enter a priority encoder that calculates the number of the highest-priority eligible task. When a running task invokes a task switch, control will

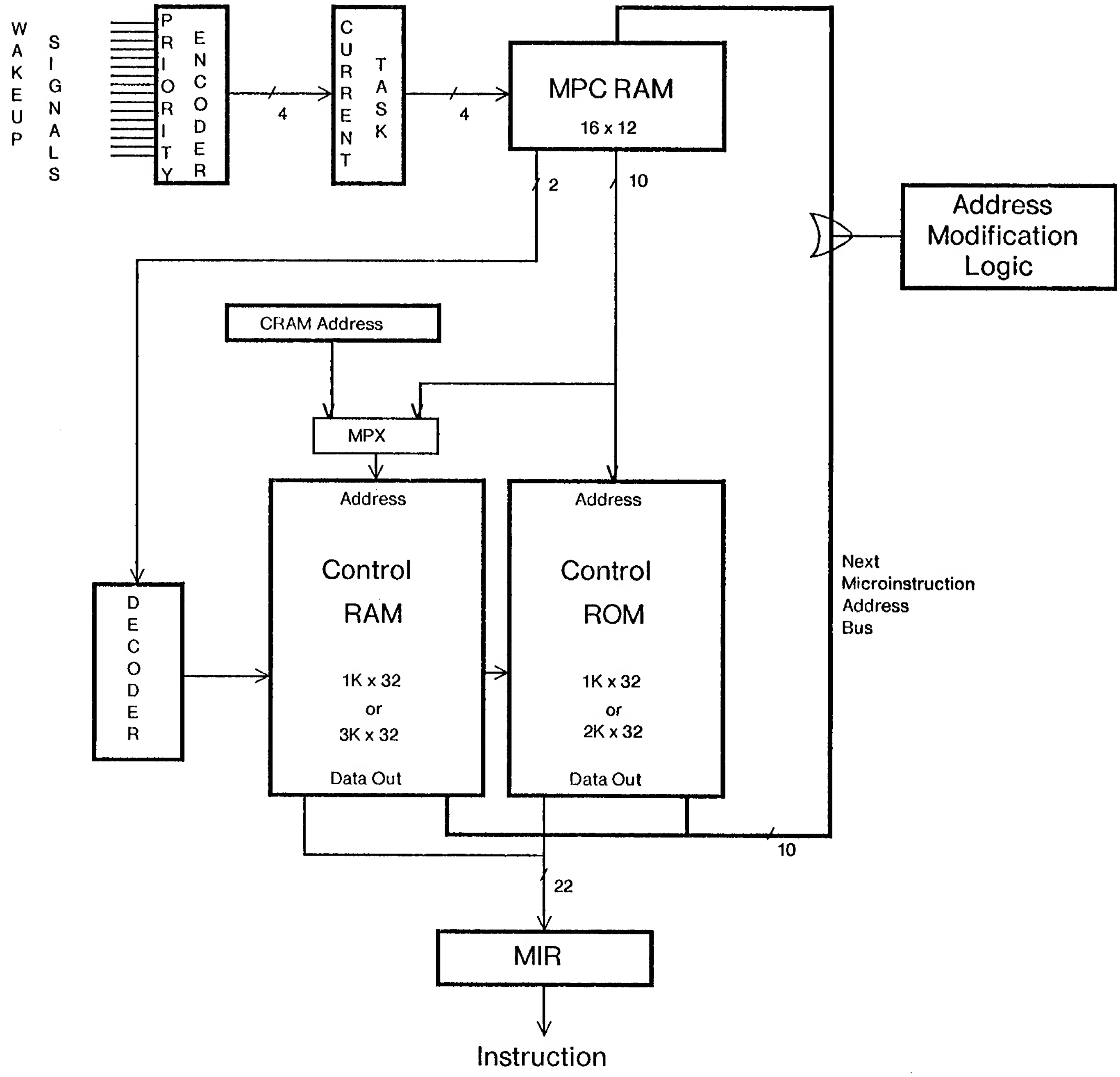


Figure 2 -- Processor Control

switch to another task only if a higher priority task has a wakeup signal held true, or if the current task no longer has a wakeup signal true. In the latter case, control goes to a lower priority task. The lowest priority task is the CPU emulator, which is always requesting wakeup.

If the processor executes the TASK function (F1=2) during an instruction, the current task register is loaded (at the end of the instruction) with the number of the highest priority task currently requesting a wakeup. This causes the next instruction to be fetched from the ROM location specified by the saved task's MPC. One additional instruction is executed by the current task before the switch becomes effective. This instruction may execute task-specific functions, ***but it must do no NEXT address modification***, since any such modification would affect the new task. The situation for two streams of instructions A-F and J-M in two different tasks is shown below:

Instruction being executed	Instruction being fetched	Address stored in MPC at end of cycle
A	B	C
B	C	D
C[1]	D	E
D	J	K
J[2]	K	L
K[3]	L	M
L	E	F
E	F	G

[1]Instruction C allows task switching. New task's MPC = J.
[2]Instruction J does an operation which removes its task's wakeup request.
[3]Instruction K allows task switching, and the original task is now highest priority.

The BLOCK function (F1=3) is used, by convention, to signal a hardware device associated with the currently running task to remove its wakeup signal. This function is *not* accomplished by the Alto microprocessor, but rather by the individual device interfaces.

Task switches must occur only at times when the current task has no state in any register (except R registers dedicated to the task) and has no main memory operation in progress, since there is no provision in the hardware for saving this information. That is, all state important to the task must have been stored in safe places by the end of the microinstruction after the one containing the TASK function. It is not legal to place TASK functions in two consecutive microinstructions.

INITIALIZATION

The only way in which the microprogram can affect the task structure is to request a task switch. In particular, it cannot affect the MPCs of tasks other than itself. This presents an initialization problem which is solved by having each task start at the location which is its task number (thus the emulator task finds its first instruction to execute at MPC=0). Task numbers are written into the MPC RAM during a reset cycle, which may be initiated manually or by a CPU instruction (see SIO instruction in section 3.3). Tasks ordinarily begin execution in ROM0. In order to start tasks in the RAM, there is a mechanism for modifying the initial MPC's of tasks so that they will begin execution in RAM0 (see section 8.4)

STANDARD TASKS

The standard Alto and its associated device controllers use many of the available tasks. Detailed descriptions of the operation of most tasks are found in the sections of this manual relevant to the hardware devices. Appendix D is a list of the standard tasks.

3.0 EMULATOR

The lowest-priority Alto task is called the Emulator task. This task is always requesting wakeup, but can be interrupted by a wakeup request from any other task. In effect, the emulator task is the "background job." The standard Alto microcode ROM includes standard emulator task microcode for fetching from Alto memory, decoding, and interpreting instructions from the Standard Instruction Set. In the rest of this chapter we shall frequently use the term "emulator" to mean "standard emulator task microcode." This standard microcode can be extended or replaced, usually by executing special emulator task microcode in the microinstruction RAM.

This section describes microcode versions installed after June 1976. To determine the vintage of a machine's microcode, see descriptions of SIO and VERS (section 3.2).

3.1 Standard Instruction Set

REGISTERS

The emulator state is carried from instruction to instruction in several registers:

PC: The "program counter," which contains the 16-bit address of the next instruction to be fetched and executed. It is actually implemented as R-register 6.

AC0, AC1, AC2, AC3: The accumulators, each of which contains 16 bits. Instructions are available for transferring contents of accumulators to and from memory registers and for performing arithmetic and logical operations among accumulators. The notation AC(n) is often used to refer to the contents of accumulator n (n=0,1,2,3). These accumulators are implemented as R-registers 3-0 respectively.

C: The "carry" bit which is modified by most arithmetic operations. It is implemented as special hardware (see section 3.5).

MEMORY: The Alto has "64K" 16-bit memory words, addressed by values ranging from 0 to 176777B. Addresses 177000B to 177777B are reserved for various I/O device uses (see Appendix B). Memory on Alto IIs can be extended to 256K in 64K banks (see Section 2.3).

Additional R- and S-registers may be used temporarily during emulation of a single instruction.

INSTRUCTION FORMAT

The standard instruction set is best described by breaking it into four groups according to the way the instructions are formatted (see Figure 3).

Several of the instructions compute an "effective address" based on the values of the I (indirect), X (index) and DISP (displacement) fields of the M-group, J-group and some S-group instructions. The effective address calculation is best described by a brief "program." First we define the function SignExtend(x) to represent the sign-extension of the 8-bit number x:

SignExtend(x) = if x $\geq$ 200B then x+177400B else x.

Then EffAddr(), the function to compute the effective address is:

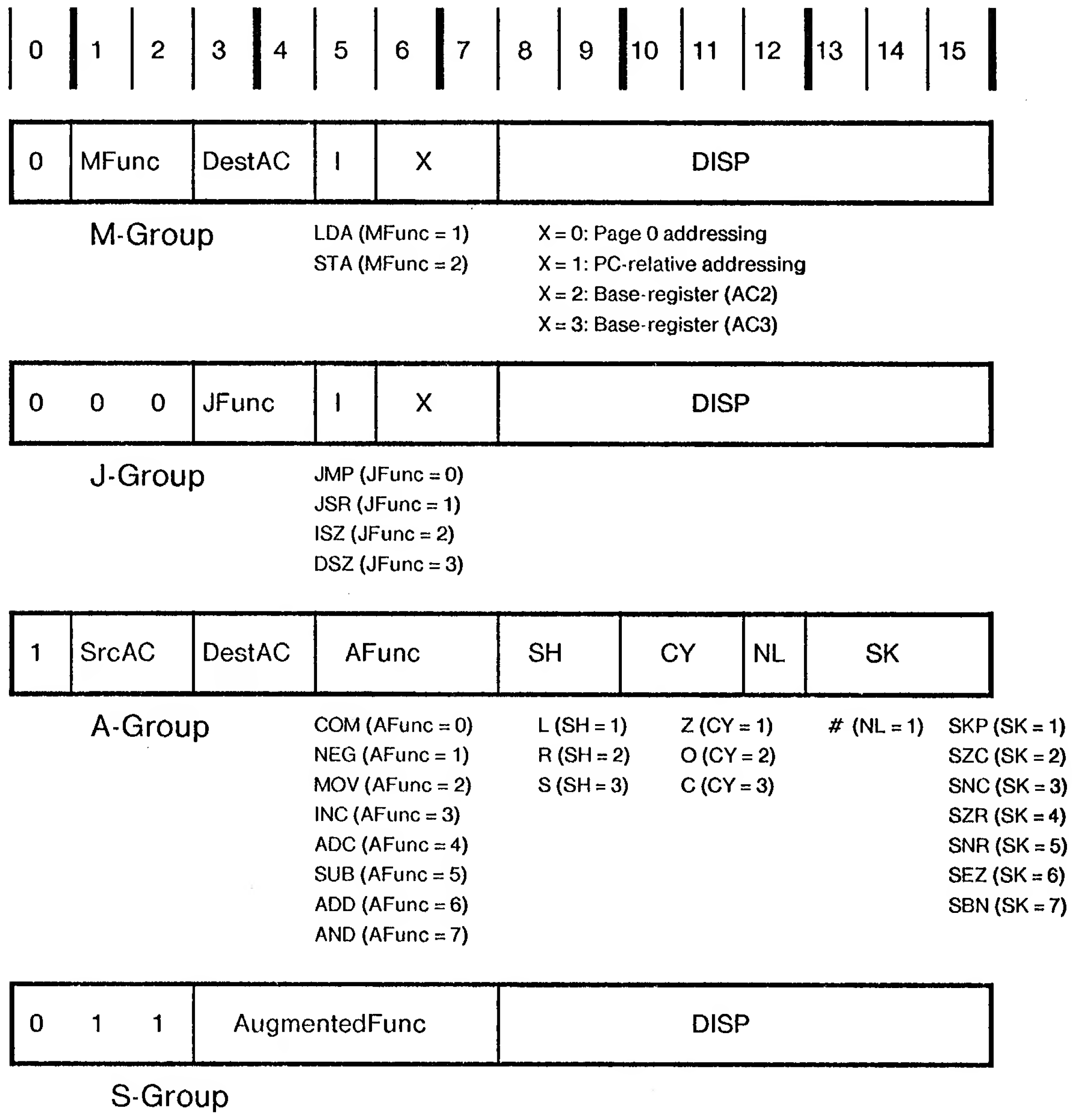


Figure 3 -- Instruction Formats

```
EffAddr() =
[                                   //The symbol "E" denotes effective address
E ←   (                             //Values of I,X, and DISP are from the instruction
      if      X=0 then DISP                       //"page 0 addressing"
      elseif  X=1 then SignExtend(DISP)+PC        //"relative addressing"
      elseif  X=2 then SignExtend(DISP)+AC(2)     //"base register addressing"
      elseif  X=3 then SignExtend(DISP)+AC(3)     //"base register addressing"
      )
if I ≠ 0 then E←rv(E)                           //Now do single-level indirection
].
```

The notation for these addressing modes is demonstrated below. The DISP value is always specified first; the X value is not given explicitly, but is determined either by the address of the label or by a modifier ",2" or ",3" which specifies base register indexing:

```
JMP LABEL2          ; If LABEL2 is in page 0, X=0; otherwise X=1.
JMP 15,3            ; DISP=15; 3 means use AC3 as base register.
JMP @3              ; The character @ causes I to be 1.
```

Note that instructions which compute an effective address always do so before any other operations. Thus JSR 1,3 computes the effective address of 1+AC(3) before saving PC+1 in AC3.

MEMORY GROUP OPERATIONS

The DestAC field specifies one of the four accumulators (DestAC=0 for AC0, DestAC=1 for AC1, etc.). The MFunc field specifies one of two operations:

Mnemonic	MFunc	Action
LDA	1	This operation loads an accumulator from memory. AC(DestAC)←rv(E).
STA	2	This operation stores an accumulator into memory. rv(E)←AC(DestAC).

These instructions are written by giving the mnemonic, followed by the accumulator number (DestAC), followed by an effective address notation:

```
STA 3 .+4           ; Store AC3 in the fourth location following here
LDA 0 4,2           ; Load AC0 from address=4+AC(2)
LDA 0 @.+2          ; Load AC0 from address contained in second location following here
```

JUMP AND MODIFY GROUP OPERATIONS

The JFunc field specifies one of four operations:

Mnemonic	JFunc	Action
JMP	0	This operation causes a "jump" by changing the value of the PC. PC←E.
JSR	1	This operation is useful when calling subroutines because it saves a return address in AC3. AC(3)←PC+1; PC←E.
ISZ	2	This operation increments the contents of a memory cell and skips if the new contents are zero. rv(E)←rv(E)+1; if rv(E)=0 then PC←PC+1. This instruction does not alter the C bit.
DSZ	3	This instruction decrements the contents of a memory cell and skips if the new contents are zero. rv(E)←rv(E)-1; if rv(E)=0 then PC←PC+1. This instruction does not alter the C bit.

These instructions are written by giving the mnemonic and the effective address notation:

```
JSR SUBR        ; AC3 is left pointing to the location after this one
JMP 1,3         ; Jump to AC(3)+1
```

ARITHMETIC GROUP OPERATIONS

All 8 of these instructions operate on the contents of the accumulators and the carry bit. Typically, a binary operation involves the contents of the "source accumulator" (SrcAC) and the "destination accumulator" (DestAC) and leaves the result in the destination accumulator. The carry bit (C bit) and the PC can also be modified in the process.

The operation of the instructions is best explained by following the flow in Figure 4. The 16-bit contents of the source and destination accumulators are fetched and passed to the function generator.

The carry generator produces an output that depends on the value of the C bit and the CY field of the instruction:

Mnemonic	CY	Output
none	0	C
Z	1	0
O	2	1
C	3	1-C (i.e., the complement of C).

The function generator is controlled by the AFunc field; various values will be described below. It takes two 16-bit numbers and a carry input and generates a 16-bit Result and a carryResult.

The shifter is controlled by the SH field in the instruction:

Mnemonic	SH	Action
none	0	No shifting; the 17 output bits are the same as the 17 input bits.
L	1	Rotate the 17 input bits left by one bit. This has the effect of rotating bit 0 left into the carry position and the carry bit into bit 15.
R	2	Rotate the 17 bits right by one bit. Bit 15 is rotated into the carry position and the carry bit into bit 0.
S	3	Swap the 8-bit halves of the 16-bit result. The carry is not affected.

The skip sensor tests various of the 17 bits presented to it and may cause a skip (PC←PC+1) if an appropriate condition is detected:

Mnemonic	SK	Action
none	0	Never skip
SKP	1	Always skip
SZC	2	Skip if the carryResult is zero
SNC	3	Skip if the carryResult is non-zero
SZR	4	Skip if the 16-bit Result is zero
SNR	5	Skip if the 16-bit Result is non-zero
SEZ	6	Skip if either carryResult or Result is zero
SBN	7	Skip if both carryResult and Result are non-zero

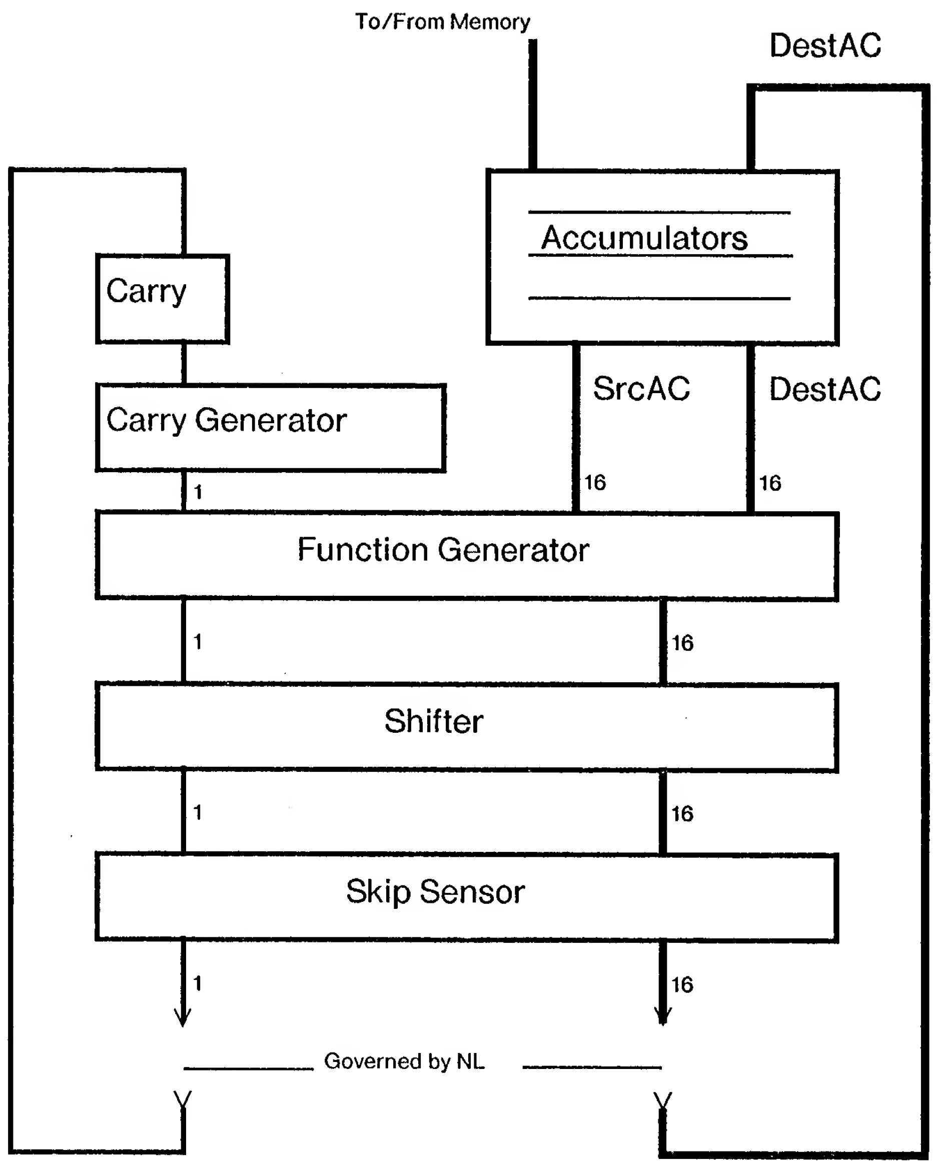


Figure 4 -- Instruction Execution

The alert reader will detect that the SK field is microcoded. The skip condition can be described as:

```
skip = (SK[2]≠0) XOR
       ((SK[0]≠0 AND result=0) OR (SK[1]≠0 AND carryResult=0))
```

where SK[0] is the first bit of the field, SK[1] the second and SK[2]the third.

The NL bit in the instruction controls the operation of the switch in the illustration. If NL=1, neither the destination accumulator nor the carry bit is loaded; otherwise the destination accumulator is loaded from Result and the carry bit from carryResult. The "no-load" feature is useful for instructions whose only use is testing some value. The character # is appended to the mnemonic for operations if the NL bit is to be set.

The AFunc operations are described below. Note that "Result" will be stored into the destination accumulator (DestAC) unless NL=1.

Mnemonic	AFunc	Operation	Description
COM	0	COMPLEMENT	The function generator produces the logical complement of AC(SrcAC). It passes the carry bit unaffected.
NEG	1	NEGATE	The function generator produces the two's complement of AC(SrcAC). If AC(SrcAC) contains zero, complement the value of the carry supplied to the function generator, otherwise supply the specified value.
MOV	2	MOVE	The function generator passes AC(SrcAC) and the carry bit unaffected.
INC	3	INCREMENT	The Result produced is AC(SrcAC)+1; the carry is complemented if AC(SrcAC)=177777B.
ADC	4	ADD COMPLEMENT	The Result produced is the sum of AC(DestAC) and the logical complement of AC(SrcAC). The carry bit is complemented if the addition generates a carry.
SUB	5	SUBTRACT	Subtracts by adding the two's complement of AC(SrcAC) to AC(DestAC). The carry bit is complemented if the addition generates a carry.
ADD	6	ADD	Adds AC(SrcAC) to AC(DestAC). The carry bit is complemented if the addition generates a carry.
AND	7	AND	The Result is the logical and of AC(SrcAC) and AC(DestAC). The carry is passed unaffected.

The arithmetic instructions are written by citing the AFunc mnemonic, followed optionally by the CY mnemonic, followed optionally by the SH mnemonic, followed optionally by the NL mnemonic. Then after a space, the source accumulator number is given, the destination accumulator number, and optionally an SK mnemonic. For example:

```
SUB 0 0           ; Zero AC0 by subtracting it from itself
MOVZ 2 1          ; Move AC2 to AC1, and zero C
SUBZL 1 1         ; Set AC1 to 1
ADC 0 0           ; Set AC0 to 177777B
SUB# 2 3 SNR      ; Skips if AC2 and AC3 are unequal but
                  ; affects neither
COM# 1 1 SZR      ; Skips if AC1 is 177777B but leaves it unchanged
SUBZ# 1 0 SZC     ; Skips if AC0<AC1 unsigned
ADCZ# 1 0 SZC     ; Skips if AC0≤AC1 unsigned
```

To subtract the constant 1 from AC1:

```
NEG 1 1
COM 1 1
```

To OR together the contents of AC0 and AC1; result in AC0:

```
COM 1 1
AND 1 0
ADC 1 0
```

To XOR together the contents of AC0 and AC1; result in AC0:

```
MOV 0 2
ANDZL 1 2
ADD 1 0
SUB 2 0
```

To negate a double-length number in AC0 and AC1:

```
NEG 1 1 SNR
NEG 0 0 SKP
COM 0 0
```

To add the double-length number in AC2,AC3 to one in AC0,AC1:

```
ADDZ 3 1 SZC
INC 2 2
ADD 2 0
```

To subtract the double-length number in AC2,AC3 from one in AC0,AC1:

```
SUBZ 3 1 SZC
SUB 2 0 SKP
ADC 2 0
```

The Bcpl construct "if a gr b then ..." uses code which does a subtract and checks the sign. Unfortunately, this is not a true signed compare because the subtract may overflow. With this code, 2 gr 0 is true, but 077777B gr 100000B is false (077777B is the largest positive number and 100000B the largest negative). The code generated by Bcpl looks like:

```
LDA 0 4,2           ; Pick up a
LDA 1 5,2           ; Pick up b
ADCL# 1 0 SZC       ; Subtract and check sign
JMP falsePart       ; Not true
JMP truePart        ; True
```

The "true signed compare" for a>b is:

```
LDA 0 4,2           ; Pick up a
LDA 1 5,2           ; Pick up b
SUBZR 2 2           ; Place 100000B in AC2
AND 1 2             ; AC2=(if b<0 then 100000B else 0)
ADDL 0 2            ; CARRY=(if a and b signs differ then 1 else 0)
ADC# 1 0 SNC
JMP falsePart
JMP truePart
```

S-GROUP INSTRUCTIONS

Opcodes in the range 60000B-77777B, are assigned to the S-group, which comprises a variety of miscellaneous instructions and unimplemented operations. Bits 3 through 7 of the instruction determine 32 opcodes, each of which may use the displacement field (bits 8-15 of the instruction). One of these opcodes (61xxx, $0 \leq xxx \leq 377B$) uses the displacement field to represent up to 256 instructions which do not require a displacement or a parameter as part of the opcode.

Currently, only a small number of the available S-group instructions have been implemented. The remaining unimplemented instructions all trap in one of two ways:

ROM trap — PC is saved in location TRAPPC, and then a JMP@ TRAPVEC+OP instruction is simulated. OP is bits 3-7 of the trapping instruction.

TRAPPC	527B	When an unimplemented opcode is executed by the emulator, the PC is saved here. It points to the location after the trapping instruction.
TRAPVEC	530B-567B	Contains pointers to the trap routines for the 32 opcodes (bits 3-7 of the trapping instruction). The first word corresponds to opcode 60xxx, 0≤xxx≤377B.

RAM trap — If no microinstruction RAM is present, the trap is handled as a ROM trap. If a RAM is present, the microcode transfers to location TRAP1 in the RAM with the trapping instruction in L, the instruction cycled by 8 bits in the R-register XREG, and PC pointing to the location after the trapping instruction.

This arrangement makes it convenient to extend the Alto's standard instruction set by implementing additional functions in software which is dispatched to via TRAPVEC, or in microcode which is dispatched to via a RAM trap. An appendix tabulates the S-group instruction set opcodes and what each does or how it traps.

MUL 61020B Unsigned multiply:

Multiply the unsigned integers in AC1 and AC2 to generate a 32-bit product; add the product to the integer in AC0. Leave the high-order part of the result in AC0 and the low-order part in AC1. AC2 is unaffected.

DIV 61021B Unsigned divide:

The double-length unsigned integer in AC0 and AC1 is divided by the unsigned integer in AC2. The quotient is left in AC1; the remainder in AC0. AC2 is unaffected. The instruction normally skips the next instruction; if overflow occurs (AC0 ≥ AC2 unsigned), DIV does not skip.

CYCLE 60000B Left cycle AC0:

Left cycle (rotate) the contents of AC0 by the amount specified in instruction bits 12-15, unless this value is zero, in which case cycle AC0 left by the amount specified in bits 12-15 of AC1.

JSRII 64400B Jump to subroutine double indirect, PC relative:

```
AC3←PC+1
PC←rv(rv(PC+SignExtend(DISP)))
```

JSRIS 65000B Jump to subroutine double indirect, AC2 relative:

```
AC3←PC+1
PC←rv(rv(AC2+SignExtend(DISP)))
```

CONVERT 67000B Scan convert a font character:

The CONVERT instruction does scan conversion of characters, i.e., it transfers data between an area of main memory containing a font and an area of memory containing a bit map to be displayed on the TV monitor.

CONVERT takes a number of arguments:

AC0 contains the address of the destination word into which the upper left corner of the character is to be placed, offset by NWRDS, the number of words to be displayed on each scan line (AC0=DWA-NWRDS).

AC3 points to a character pointer in the font for the character to be displayed (AC3=FONTBASE+CHARACTER CODE).

AC2+SignExtend(DISP) is the address of a two-word table:

word 0:	NWRDS (number of words per to scan line); NWRDS $<$ 128.
word 1:	DBA, the destination bit address corresponding to the left hand edge of the character. CONVERT interprets this bit address reversed from the normal convention, i.e., 0 is the least significant bit, 15 the most significant bit.

CONVERT requires that a 16 word mask table be set up starting at MASKTAB (460B) in page 1. rv(MASKTAB+n)=(2↑(n+1))-1 ($0 \leq n \leq 15$).

The format of an Alto font designed for use with CONVERT is given below; names of font files in this format conventionally have an extension ".AL". The CONVERT instruction does not examine the words at FONTBASE-2 and FONTBASE-1; these are provided solely for convenience of software.

FONTBASE-2:

The height of a line of text in scan lines. This number incorporates the effects of the highest and lowest character in the font, i.e. it is max(HD+XH)-min(HD) where the max and min are taken independently and HD and XH are defined below.

FONTBASE-1:

Bit 0:	0 = Fixed width font. 1 = Proportional width font.
Bits 1-7:	Baseline -- number of scan-lines from top of highest character in font to the baseline.
Bits 8-15:	The width of the widest character in raster points.

FONTBASE to FONTBASE+377B:

Self-relative pointers to word XW of the character descriptor block for the character codes 0-377B.

FONTBASE+400B to FONTBASE+400B+EXTCNT-1:

These locations contain self-relative pointers to word XW of the character descriptor blocks for extensions, i.e., portions of characters which are wider than 16 bits. EXTCNT is the total number of character extensions.

FONTBASE+400B+EXTCNT to end:

Contains a number of character descriptor blocks of the form:

word 0 to word XW-1:
The bit map for the character and surrounding spaces. The bit map does not include 0's at the top and bottom of the character, as the character will be vertically positioned by CONVERT. The upper left-hand bit of the character is in the MSB of word 0.

word XW:
If the character is $\leq$ 16 bits wide, this word contains (2*width)+1. If the character is > 16 bits wide, this word contains 2* a pseudo-character which is used as a character code to index an extension character in the font. If this is the last extension block of a character, this word contains (2* the width of the final extension), rather than the total width. The pointer indexed by the character code points to this word.

word XW+1:
In the left byte, HD. In the right byte, XH. HD is the number of scan lines to skip before displaying the character, XH is the height of the bit map for this character.

The CONVERT instruction ORs the character bitmap into the display area. If the character does not require an extension, CONVERT skips, with the following information in the AC's:

AC0:	unchanged
AC1:	DBA AND 17B
AC2:	unchanged
AC3:	the width of the character in bits

If the character requires an extension, CONVERT returns does not skip. AC3 contains the pseudo-character code for the extension, and AC's 0-2 are as above.

RCLK 61003B Read Clock:

The microcode maintains a 26 bit real time clock which is incremented by the memory refresh task at 38.08 microsecond intervals (more precisely, once every 224 ticks of the system clock, whose nominal frequency is 5.880000 MHz). The high-order 16 bits of this clock are maintained in location RTC (430B) in page 1 The low-order 10 bits are kept in R37. The remaining 6 bits of R37 contain state information unrelated to the time. RCLK loads AC0 with the contents of location RTC, and loads AC1 with the contents of R37. The period of the full 26-bit clock is about 40 minutes.

The contents of R37 are slightly different on Alto I and Alto II: on Alto I, R37[0-9] contain the low order clock bits; on Alto II, R37[4-13] are used. Consequently, on the Alto I, the contents of AC0 and AC1 returned by RCLK may be viewed as a 32-bit clock in units of .595 microseconds, provided AC1[10-15] is first zeroed.

SIO 61004B Start I/O:

Start I/O is included to facilitate I/O control. It places the contents of AC0 on the processor bus and executes the STARTF function (F1=17B). By convention, bits of AC0 must be "1" in order to signal devices. See Appendix C for a summary of assigned bits.

If bit 0 of AC0 is 1, and if an Ethernet board is plugged into the Alto, the machine will boot, just as if the "boot button" were pressed (see sections 3.4, 8.4, and 9.2.2 for discussions of bootstrapping).

SIO also returns a result in AC0. If the Ethernet hardware is installed, the serial number and/or Ethernet host address of the machine (0-377B) is loaded into AC0[8-15]. (On Alto I, the serial number and Ethernet host address are equivalent; on Alto II, the value loaded into AC0 is the Ethernet host address only.) If Ethernet hardware is missing, AC0[8-15] = 377B. Microcode installed after June 1976, which this manual describes, returns AC0[0]=0. Microcode installed prior to June 1976 returns AC0[0]=1; this is a quick way to acquire the approximate vintage of a machine's microcode.

BLT	61005B	Block transfer:
BLKS	61006B	Block store:

These instructions use tight microcode loops to move a block of memory from one place to another (BLT) or to store a constant value into a block of memory (BLKS). Block transfer and block store take the following arguments:

AC0:	Address of the first source word-1 (BLT), or data to be stored (BLKS).
AC1:	Address of the last word of the destination area.
AC3:	Negative word count.

Because these instructions are potentially time consuming, and keep their state in the AC's, they are interruptable. If an interrupt occurs, the PC is decremented by one, and the AC's contain the intermediate state. On return, the instruction continues. On completion, the AC's are:

AC0:	Address of last source word+1 (BLT), or unchanged (BLKS).
AC1:	Unchanged.
AC2:	Unchanged.
AC3:	0.

The first word of the destination area (AC1 + AC3 + 1) is the first to be stored into.

SIT	61007B	Start interval timer:

The microcode implements an interval timer which has a resolution of 38.08 microseconds, and a maximum period of 10 bits. Because the principal application for this timer is to do bit sampling for a serial EIA-RS232 compatible communications line, the timer is specialized for this purpose. It uses three dedicated locations in page 1:

ITTIME	525B	Contains the time at which the next timer interrupt should be caused. On Alto I, the 10-bit time is stored in ITTIME[0-9], and the remaining bits must be zero. On Alto II, the time is stored in ITTIME[4-13], and the remaining bits must be zero.
ITIBITS	423B	This word contains one or more bits specifying the channel or channels on which the timer interrupt is to occur.
ITQUAN	422B	When the interval timer interrupt is caused, the microcode stores a quantity in this location which depends on the mode.

The SIT instruction ORs the contents of AC0 into R37. The high 14 bits should be 0; the low-order 2 bits determine the interval timer mode:

R37[14-15]

0 Off.

1 Normal mode. Every 38.08 microseconds, test to see if (R37 AND TIMEMASK)=ITTIME (on Alto I, TIMEMASK=177700B; on Alto II, the proper value for TIMEMASK is 7774B, but version 23 of Alto II microcode uses a value of 7700B). If they are equal, cause an interrupt on the channel specified by ITBITS. Store the current state of the EIA interface in ITQUAN, and set R37[14-15] to zero. The state of the EIA interface is bit 15 of location EIALOC (177701B) in page 377B. This bit is 0 if the line is spacing, 1 if it is marking.

2 Same as 0.

3 Every 38.08 microseconds, check the state of the EIA line by reading EIALOC. If the line is marking (EIALOC is non zero), do nothing. If the line is spacing, cause an interrupt on the channel specified by ITBITS. Store the current value of R37 in ITQUAN, and set R37[14-15] to zero.

The intention is that a program which does EIA input can use mode 3 to monitor the line for the arrival of a character, and can then use mode 1 to time the center of each bit. By storing the state of the line, the interrupt latency can be as much as 1 bit time without errors.

JMPRAM 61010B Jump to RAM: (see section 8.5 for details)

Switches the emulator task micro PC to another microinstruction bank in ROM or RAM The next emulator microinstruction will be determined from the value in AC1 (mod 1024) -- see the discussion of bank switching in section 8.4.

RDRAM 61011B Read RAM: (see section 8.5 for details)

Reads the control RAM halfword addressed by AC1 into AC0.

Note: In Alto IIs running microcode version 2, this instruction does not work reliably if the Ethernet interface is running.

WRTRAM 61012B Write RAM: (see section 8.5 for details)

Writes AC0 into the high-order half and AC3 into the low-order half of the control RAM word addressed by AC1.

VERS 61014B Version:

AC0 is loaded with a number which is coded as follows:

bits 0-3 Alto engineering number

0 or 1	Alto I
2	Alto II
3	Alto II with extended memory

bits 4-7 Alto build number.

bits 8-15 Version number of the microcode.

This instruction permits programs to know the differences among various kinds of Altos. Use of the Alto build number (bits 4-7) has been abandoned; its contents are undefined. The two flavors of Alto maintain separate enumerations of microcode versions (see section 9 for some conventions).

DREAD 61015B Double-word read (Alto II only):

```
AC0← rv(AC3); AC1← rv(AC3 XOR 1)
```

DWRITE 61016B Double-word write (Alto II only):

```
rv(AC3)← AC0; rv(AC3 XOR 1)←AC1
```

DEXCH 61017B Double-word exchange (Alto II only):

```
t← rv(AC3); rv(AC3)← AC0; AC0←t
t← rv(AC3 XOR 1); rv(AC3 XOR 1)← AC1; AC1← t
```

DIAGNOSE1 61022B Diagnostic instruction (Alto II only):

This instruction starts a special double-word write cycle that also writes the Hamming code check bits.

```
rv(177026B)← AC2 (set Hamming code)
rv(AC3)← AC0; rv(AC3 XOR 1)← AC1
```

DIAGNOSE2 61023B Diagnostic instruction (Alto II only):

This instruction writes the same memory location with two different values in quick succession:

```
rv(AC3)← AC0
rv(AC3)← AC0 xor AC1
AC0← AC0 xor AC1
```

BITBLT 61024B Bit-boundary block transfer:

An instruction for moving bits around in memory. It is particularly helpful for dealing with the display bit map. BITBLT requires the RAM to be present in order to use some S registers (41B through 51B). If the RAM is not present, BITBLT will trap as if it were an unimplemented operation.

CALLING SEQUENCE

The BITBLT function is invoked with:

AC1:	0
AC2:	pointer to BBTable, which must be even.

Only AC2 is preserved by BITBLT.

The most common errors when using this instruction are failing to align the BBTable on an even word boundary, failing to zero AC1, and failing to zero FUNCTION[0-9].

The format of the BBTable is:

Word	Name	Remarks
0	FUNCTION	Operation, SourceType, Bank, etc
1	unused	
2	DBCA	Destination BCA
3	DBMR*	Destination BMR
4	DLX*	Destination LX
5	DTY*	Destination TY
6	DW*	Destination W
7	DH*	Destination H
8	SBCA	Source BCA
9	SBMR	Source BMR
10	SLX*	Source LX
11	STY*	Source TY
12	Gray0	Four words to specify gray block...
13	Gray1	
14	Gray2	
15	Gray3	

*These should all be positive values, although if DH<0 or DW<0 then BITBLT is a NOP.

Trick: since BITBLT uses all of the accumulators, BCPL programmers must save AC2, the stack pointer, somewhere. Put it in word 1 of the BBTable, since AC2 still points at the table after the instruction finishes, making it easy to recover.

The instruction is interruptable as it begins consideration of each scan line. If an interrupt happens, the state of its progress is saved in AC1 and the PC is backed up so that on return from the interrupt, BITBLT will finish its job. This is the reason why AC1 must be zero when starting the instruction.

DEFINITIONS

A *bit map* is a region of memory defined by BCA and BMR, where BCA is the *base core address* (starting location) and BMR is the *bit map raster width* in words; the number of scan lines is irrelevant for our purposes. (If both BMR and BCA are even, then the bit map may be displayed on the screen using standard Alto facilities.)

A *block* is a rectangle within a bit map. It has four corners which need not fall on word boundaries. A block is described by 6 numbers:

BCA	Bit map's base core address
BMR	Bit map's width in words
LX	Block's left X ("x offset" from first bit of scan-line)
TY	Block's top Y ("y offset" from first scan-line)
W	Block's width in bits
H	Block's height in scan-lines

Example: A block is used to designate a sequence of bits in memory, such as a 16 wide 14 high region containing the bit pattern of a font character. In this case, BCA points to the font character, BMR is 1, LX and TY are 0, W is 16, and H is 14. If source and destination blocks overlap, they had better have the same BCA.

BLOCK OPERATIONS

The basic block operations operate by storing some bits into a "destination block." The source of these bits varies; often it is another block, the "source block." There are various functions that BITBLT can perform.

The FUNCTION word of the BBTable contains a number of fields:

FUNCTION[0-9]	Must be zero
FUNCTION[10]	Source block is in the alternate bank
FUNCTION[11]	Destination block is in the alternate bank
FUNCTION[12-13]	SourceType
FUNCTION[14-15]	Operation

The *operation* field specifies the operation to be performed on the source and destination blocks:

Operation	Name	Action
0	Replace	Destination Block ← *Source*
1	Paint	Destination Block ← *Source* OR *Destination*
2	Invert	Destination Block ← *Source* XOR *Destination*
3	Erase	Destination Block ← (NOT *Source*) AND *Destination*

The *SourceType* specifies how the *Source* as used in the above 4 operations is to be computed. The encodings are:

SourceType	Meaning
0	*Source* is a block of a bit map
1	*Source* is the complement of a block of a bit map
2	*Source* is the logical "and" of a source block and the "gray block" (see below).
3	*Source* is the "gray block."

The "gray block" is conceptually a block of infinite extent in which a pattern of dots is repeated. The pattern is specified by four words (Gray0 through Gray3). These give the patterns to write into the destination block where called for, one gray word per scan line. The words will align with destination block word boundaries, but BITBLT will use Gray0 through Gray3 in the order in which BITBLT processes scanlines (either top to bottom (DTY<STY) or bottom to top (DTY≥STY)).

The most common use of these gray values is to generate a uniform pattern. While the BITBLT instruction takes care of going through these values appropriately, the table must be phased properly to eliminate *seams*. Specifically, if A B C D are the desired 16-bit word-aligned values of gray for scan-lines 0 1 2 3 (mod 4), then two adjustments must be made:

```
Let Q = DTY + 1.
If DTY < STY, then exchange B and D and let Q = -(DTY+DH+2).
Rotate the pattern left (i.e., A←B, B←C, etc) a total of (Q AND 3) times.
Set Gray0←A, Gray1←B, Gray2←C, Gray3←D
```

When the source is a block of bit map, the width and height parameters of the block are not needed: the width and height of the destination block are also used as the width and height of the source block. It is permissible for the source and destination blocks to overlap, such as when sliding an existing block around within a bit map; BITBLT will move words in the order required for the correct results. However, if the source and destination blocks do overlap, they must belong to the same bit map (i.e., DBCA=SBCA and DBMR=SBMR).

TIMING DETAILS

The microcode has roughly the following speed characteristics:

Horizontally, along one raster line (so to speak):

store constant	13 cycles/word
move block	23 cycles/word
if skew not zero	add 6
if source not zero	add 7
1st or last word	add 13
function not store	add 6

Vertical loop overhead (time to change raster lines):

14-21 cycles/scanline, depending on source/dest alignment
add 6 if function uses gray

Initial setup overhead (time to start or resume from interrupt):

approximately 240 cycles

Total for a typical character, 8 wide by 14 high:

approximately 1500 cycles

These timings all in units of Alto microinstruction cycles and *do* include all memory wait time and *do not* include any degradation due to competing tasks, such as the display or disk. For typical characters on the Alto screen, BITBLT is about 2/3 the speed of CONVERT.

XMLDA 61025B Extended Memory Load Accumulator (Alto II only)

Loads AC0 from the location addressed by AC1 in the alternate bank.

XMSTA 61026B Extended Memory Store Accumulator (Alto II only)

Stores AC0 into the location addressed by AC1 in the alternate bank. If the the addressed bank of memory has not been installed, the instruction yields undefined results and will probably cause a parity error. See section 2.3.

3.2 Interrupts

The emulator microcode provides 15 channels of vectored interrupts. The microcode implements only a single level of interrupts; however, a multi-level priority interrupt system may easily be implemented in software (see below).

Interrupts may be caused in two ways:

microcode	This method is used by I/O device microcode. A device usually has a dedicated location in which the CPU program places a word containing ones in the bit positions corresponding to the channels on which to cause interrupt(s) upon completion of I/O activity. The emulator is guaranteed to notice an interrupt caused in this way within one instruction.
software	This method is used by a CPU program. A program causes interrupts by ORing into location WW one bits corresponding to the channels on which interrupts should occur. The emulator is *not* guaranteed to notice an interrupt caused in this way until an EIR instruction is executed.

When an interrupt occurs, further interrupts are disabled and the state of the interrupted CPU program is contained in AC0-3, CARRY, and PC, which must be saved and restored by the interrupt routine. Interrupts can occur between instructions or during long instructions, in which case the instruction's intermediate state is saved in the accumulators and PC is backed up so that the interrupted instruction is re-executed when the interrupt is dismissed.

If two interrupts are requested simultaneously, the one with the highest-numbered channel will be serviced first.

The interrupt system uses a number of fixed locations in page 1:

ACTIVE	453B	This word contains ones for the channels on which interrupts are permitted to occur. Bit N is set to one to enable channel N. Bit 0 is reserved and should not be set by any program.
WW	452B	This word contains bits for channels on which interrupts are pending. This information is only valid while the interrupt system is enabled. Bit conventions are the same as for ACTIVE. WW is *not* updated when interrupts are disabled -- wakeups caused from microcode accumulate in NWW until interrupts are enabled.
PCLOC	500B	When an interrupt is initiated, the PC is saved here. If the CPU program allows nested interrupts, this location must be saved before re-enabling interrupts.
INTVEC	501B-517B	Contains pointers to the service routines for the 15 interrupt channels. The first word corresponds to channel 15 (bit 15) and the last corresponds to channel 1 (bit 1). Channel 15 is permanently assigned to handling main memory parity errors.

The interrupt system uses four instructions:

DIR 61000B Disable interrupts:

Disables the interrupt system. If more than one interrupt is initiated on a channel while interrupts are disabled, only one will occur when interrupts are re-enabled.

DIRS 61013B Disable interrupts and skip if on:

Disables the interrupt system and skips the next instruction if interrupts were enabled at the start of this instruction.

EIR 61001B Enable interrupts:

Enables the interrupt system. Interrupts initiated while interrupts were disabled occur after this instruction.

BRI 61002B Branch and return from interrupt:

Simulates a JMP @PCLOC instruction, and then enables the interrupt system. Interrupts initiated while interrupts were disabled occur after this instruction.

EXAMPLES

The code below is a sample interrupt handler for one channel, say channel 10. It permits nested interrupts from higher priority channels, where the priority is determined by software. This is accomplished by turning off all lower-priority channels and re-enabling interrupts (which were disabled by the microcode at the onset of this interrupt). Before dismissing the interrupt, it is necessary to disable the interrupt system and turn the lower-priority channels back on.

```
Interrupt:  STA 0 SavedAC0       ; save the interrupted program state
            STA 1 SavedAC1
            STA 2 SavedAC2
            STA 3 SavedAC3
            MOVR 0 0
            STA 0 SavedCarry
            LDA 0 @PCLOC
            STA 0 SavedPC

            LDA 0 @ACTIVE        ; disable lower priority channels
            STA 0 SavedActive
            LDA 1 ChanMask
            AND 1 0
            STA 0 @ACTIVE

            EIR                  ; re-enable interrupts
            ...                  ; service the interrupt
            DIR                  ; disable interrupts

            LDA 0 SavedActive
            STA 0 @ACTIVE        ; re-enable lower priority channels

            LDA 0 SavedPC        ; restore the interrupted program state
            STA 0 @PCLOC
            LDA 0 SavedCarry
            MOVL 0 0
            LDA 3 SavedAC3
            LDA 2 SavedAC2
            LDA 1 SavedAC1
            LDA 0 SavedAC0
            BRI                  ; dismiss the interrupt

SavedAC0:   0                    ; these locations must be private to this channel
SavedAC1:   0
SavedAC2:   0
SavedAC3:   0
```

```
SavedCarry:   0
SavedPC:      0
SavedActive:  0

PCLOC:        500
ACTIVE:       453
ChanMask:     37                   ; contains ones for higher priority channels
```

It is customary (though not essential) to assign interrupt channel priorities such that channel 15 has the highest priority and channel 1 the lowest. In this case, the ChanMask for channel *i*'s interrupt routine will consist of 15-*i* one bits right-justified. In any case, ChanMask *must* contain zero in the bit corresponding to the interrupt channel being serviced.

The code below initiates interrupts on the channels corresponding to one bits in AC0. It must disable interrupts to prevent WW from being changed by microcode-initiated interrupts.

```
CauseInt:     COM 0 0
              DIR
              LDA 1 @WW
              AND 0 1
              ADC 0 1              ; AC1 ← AC0 OR AC1
              STA 1 @WW
              EIR                  ; the interrupt happens after this

WW:           452
```

If a channel's ACTIVE bit is 0 when viewed from non-interrupt level, then the channel is not in use. The code below searches ACTIVE for the highest priority free channel. It is careful not to assign the parity interrupt channel. It then initializes an interrupt handler on that channel and returns a word with a one in the bit position of the assigned channel. It must not be called from interrupt level.

```
; enter with AC0 = the address of the interrupt handler
InitChan:     STA 0 INTHANDLER

              SUB 1 1              ; AC1 ← 0
              SUBZL 0 0            ; AC0 ← 1
              LDA 2 @ACTIVE
FFC:          MOVZL 0 0 SZC
               JMP fail            ; no interrupt channels free.
              INC 1 1
              AND# 0 2 SZR         ; free?
               JMP FFC             ; no.  Try the next one

              LDA 2 INTVEC         ; install handler in INTVEC
              ADD 1 2
              LDA 3 INTHANDLER
              STA 3 0 2

              LDA 2 @ACTIVE        ; turn on the channel
              ADD 0 2              ; cant carry: equivalent to OR
              STA 2 @ACTIVE
; AC0 = one-bit mask designating the assigned channel

INTVEC:       501
INTHANDLER:   0                    ; temp
```

The code below destroys the interrupt channels corresponding to one bits in AC0. It must not be called from interrupt level.

```
DestroyInt:   COM 0 0
              LDA 1 @ACTIVE
              AND 0 1
```

```
        STA 1 @ACTIVE
```

IMPLEMENTATION

In addition to the main memory locations, the interrupt system uses one R-register: NWW, new interrupts waiting. Bit 0 of NWW is 0 if the interrupt system is enabled and one if it is disabled. This is why there are only 15 channels of interrupts and why WW[0] should never be set. I/O device microcode ORs bits into this register to cause interrupts. (NWW OR WW) expresses all pending interrupts.

The main loop of the emulator checks NWW during the fetch of each emulated instruction. If NWW is greater than zero (i.e., NWW[0] is *not* set meaning the interrupt system is on, and at least one bit *is* set in NWW[1-15] meaning an interrupt is pending on some channel) then the microcode computes (NWW OR WW) AND ACTIVE. If this quantity is nonzero (i.e., an interrupt is pending and its channel is active) then an interrupt is caused. If not, NWW OR WW is stored in WW, NWW is zeroed, and the instruction is restarted.

If an interrupt is caused, the microcode stores the program counter in PCLOC, sets NWW[0] to disable further interrupts, clears the bit in NWW and in WW corresponding to the channel on which the interrupt is occurring, and loads PC with rv(INTVEC+15-CHANNEL).

When the interrupt system is disabled (by executing DIR or DIRS or initiation of an interrupt), the microcode sets NWW[0]. When the interrupt system is enabled (by executing EIR or BRI), the microcode clears NWW[0] and ORs WW into NWW.

This organization is optimized to minimize the cost (in additional microinstructions in the emulator main loop) of the most common case where the interrupt system is enabled and no interrupts are pending. When a bit appears in NWW while the interrupt system is active, it is either cleared by causing an interrupt or flushed into WW where it is checked less often, since the cost of deciding that an interrupt is pending but that the channel is inactive is too high to tolerate on each pass through the main loop. The assumption in flushing inactive bits into WW is that the CPU program will enable interrupts shortly after changing ACTIVE, and doing so will cause the pending bits in WW to be reconsidered.

3.3 Bootstrapping

The emulator contains microcode for initializing the Alto in certain ways, and thereby "bootstrapping" a runnable program into the machine. A "boot," which is invoked either by pressing the small button at the rear of the keyboard or by executing an appropriate SIO instruction (see section 3.3), simply resets all micro-PC's to fixed initial values determined by their task numbers. Unless the Reset Mode Register specifies otherwise (see section 8.4), the emulator task is started in the PROM and performs a number of operations:

1. The current value of PC is stored in memory location 0. The emulator accumulators are not altered during booting.
2. The display is turned off; i.e. rv(420B)←0.
3. Interrupts are disabled.
4. The first keyboard word (KBDAD, 177034B) is read to determine what sort of boot is to be done:

Disk Boot: If the <BS> key is not depressed, the microcode interprets any depressed keys reported in this keyboard word as a real disk address. If no keys are depressed, this results in a real disk address of 0.

The single disk sector at the given address is read: the 256 data words are read into locations 1 to 400B inclusive; the label is read into locations 402B to 411B inclusive. When the transfer is complete, PC←1, and the emulator is started. The disk status is stored in location 2, so the bootstrapping code must skip this location.

Ether Boot: If the <BS> key is depressed, the microcode anticipates breathing life into the Alto via the Ethernet. The Ethernet hardware is set up to read any packet with destination Alto number 377B into locations 1 to 400B inclusive. If a packet arrives with good status and with memory location 2 (i.e., the second word of the packet) equal to 602B (a "Breath-of-Life" packet), PC←3, and the emulator is started.

More information regarding boot loaders and boot file formats is found with Buildboot documentation in the Alto Subsystems Manual.

3.4 Hardware

There is a small amount of special hardware which is used exclusively by the emulator. This hardware is controlled by the task specific F2's, and by the ←DISP bus source.

The IR register is used to hold the current instruction. It is loaded with IR← (F2=14B). IR← also merges bus bits 0,5,6 and 7 into NEXT[6-9], which does a first level instruction dispatch.

The high order bits of IR cannot be read directly, but the displacement field of IR (8 low order bits), may be read with the ←DISP bus source. If the X field of the instruction is zero (i.e., it specifies page 0 addressing) then the DISP field of the instruction is put on BUS[8-15] and BUS[0-7] is zeroed. If the X field of the instruction is nonzero (i.e. it specifies PC-relative or base-register addressing) then the DISP field is sign-extended and put on the bus.

```
BUS[8-15]← IR[8-15]
BUS[0-7]← if IR[6-7]=0 then 0 elseif IR[8]=0 then 0 else -1
```

There are two additional F2's which assist in instruction decoding, IDISP and ←ACSOURCE. The IDISP function (F2=15B) does a 16 way dispatch under control of a PROM and a multiplexer. The values are tabulated below:

Conditions		ORed onto NEXT	Comment
if	IR[0] = 1	then 3-IR[8-9]	complement of SH field of IR
elseif	IR[1-2] = 0	then IR[3-4]	JMP, JSR, ISZ, DSZ
elseif	IR[1-2] = 1	then 4	LDA
elseif	IR[1-2] = 2	then 5	STA
elseif	IR[4-7] = 0	then 1	
elseif	IR[4-7] = 1	then 0	
elseif	IR[4-7] = 6	then 16B	CONVERT
elseif	IR[4-7] = 16B	then 6	
else		IR[4-7]	

←ACSOURCE (F2=16B) has two roles. First, it replaces the two-low order bits of the R select field with the complement of the SrcAC field of IR, (IR[1-2] XOR 3), allowing the emulator to address its accumulators (which are assigned to R0-R3). Second, a dispatch is performed:

Conditions		ORed onto NEXT	Comment
if	IR[0]=1	then 3-IR[8-9]	the complement of the SH field of IR
elseif	IR[1-2] ≠ 3	then IR[5]	the Indirect bit of IR
elseif	IR[3-7] = 0	then 2	CYCLE
elseif	IR[3-7] = 1	then 5	RAMTRAP
elseif	IR[3-7] = 2	then 3	NOPAR -- parameterless opcode group
elseif	IR[3-7] = 3	then 6	RAMTRAP
elseif	IR[3-7] = 4	then 7	RAMTRAP
elseif	IR[3-7] = 11B	then 4	JSRII
elseif	IR[3-7] = 12B	then 4	JSRIS
elseif	IR[3-7] = 16B	then 1	CONVERT
elseif	IR[3-7] = 37B	then 17B	ROMTRAP -- used by Swat, the debugger
else		16B	RAMTRAP

ACDEST, F2=13B, causes (IR[3-4] XOR 3) to be used as the low-order two bits of the RSELECT field. This addresses the accumulators from the destination field of the instruction. The selected register may be loaded or read.

The emulator has two additional bits of state, the SKIP and CARRY flip flops. CARRY is distinct from the microprocessor's ALUC0 bit, tested by the ALUCY function. CARRY is set or cleared as a function of IR and many other things (see section 3.1) when the DNS← (do novel shifts, F2=12B) function is executed. In particular, if IR[12] is true, CARRY will not change. DNS also addresses R from (3-IR[3-4]), causes a store into R unless IR[12] is set, and sets the SKIP flip flop if appropriate (see section 3.1). The emulator microcode increments PC by 1 at the beginning of the next emulated instruction if SKIP is set, using BUS+SKIP (ALUF=13B). IR← clears SKIP.

Note that the functions which replace the low bits of RSELECT with IR affect only the selection of R; they do not affect the address supplied to the constant ROM.

Two additional emulator specific functions, BUSODD (F2=10B) and MAGIC (F2=11B), are not peculiar to emulation, but are included for their general usefulness. BUSODD merges BUS[15] into NEXT[9]. MAGIC is a modifier applied to L LSH 1 and L RSH 1 to allow double length shifts. L LSH 1 and L RSH 1 normally shift zero into the vacated bit position in the shifter output. MAGIC places the high order bit of T into the low order bit of the shifter output on left shifts, and places the low order bit of T into the high order bit position of the shifter output on right shifts. (The microassembler accepts L MLSH 1 to specify the combination of L LSH 1 and MAGIC, and similarly for L MRSH 1.)

The STARTF function (F1=17B) is generated by the SIO instruction, and is used to define commands for I/O hardware, including the Ethernet.

The RSNF function (F1=16B) is decoded by the Ethernet interface, which gates the host address wired on the backplane onto BUS[8-15]. BUS[0-7] is not driven and will therefore be -1. If no Ethernet interface is present, BUS will be -1.

4.0 DISPLAY CONTROLLER

4.1 Programming Characteristics

The display controller handles transfers between the main memory and the CRT. The CRT is a standard 875 line raster-scanned TV monitor, refreshed at 60 fields per second from a bit map in main memory. The CRT contains 606 points horizontally, and 808 points vertically, or 489,648 points total.

The basic way in which information is presented on the display is by fetching a series of words from Alto main memory, and serially extracting bits to become the video signal. Therefore, 38 16-bit words are required to represent each scan line; 30704 words are required to fill the screen.

The display is defined by one or more display control blocks in main memory. Control blocks (DCB's) are linked together starting at location DASTART(420B) in page 1:

DASTART:	Pointer to word 0 of the first (top on the screen) DCB, or 0 if display is off.
DASTART+1:	Vertical field interrupt bit mask. Every 1/60 second, this word is OR'ed into NWW to cause interrupts, even if the display is off (i.e., rv(DASTART)=0).

Display control blocks must begin at even addresses in memory, and have the following format:

DCB:	Pointer to next DCB, or 0 if this is the last.	
DCB+1:	Bit 0:	0 = high resolution mode 1 = low resolution mode
	Bit 1:	0 = black on white background presentation 1 = white on black background
	Bits 2-7	(HTAB): On each scan line of this block, wait 16*HTAB bits before displaying information from memory.
	Bits 8-15	(NWRDS): Each scan line in this block is defined by NWRDS 16 bit words. (NWRDS must be even). In order to skip space on the screen without requiring bit-map, set NWRDS to 0.
DCB+2 (SA):	Bit map starting address, which must be even.	
DCB+3 (SLC):	This block defines 2*SLC scan lines, SLC in each field.	

At the start of each field, the display controller inspects DASTART and DASTART+1. An interrupt is initiated on the channel(s) specified by the bit(s) in DASTART+1. The controller then executes each DCB sequentially until the display list or the field ends. At normal resolution, the first scan line of the first (even) field of a block is taken from location SA to SA+NWRDS-1, the first scan line of the odd field is taken from locations SA+NWRDS to SA+2*NWRDS-1. During each display field, the bit map address is incremented by an extra NWRDS between each pair of scan lines. In low resolution mode, the video is generated at half speed, and each scan line is displayed twice (once in each field). During each field, the bit map address is not incremented by an extra NWRDS between the display of adjacent scan lines. This makes the format of the bit map in memory identical for both modes--only the size of the presentation is affected by the mode.

4.2 Hardware

The display controller consists of a sync generator, a data buffer and serializing shift register, and three microcode tasks which control data handling and communicate with the Alto program. The hardware is shown in block form in Figure 5. The 16 word buffer is loaded from the Alto bus with the DDR←

function (F2=10B, specific to the display word task DWT, illegal in an instruction which stops the clocks). The purpose of the intermediate buffer is to synchronize data transfers between the main buffer, which is synchronous with the 170ns. master clock, and the shift register, which is clocked with an asynchronous bit clock. The sync generator provides this clock and the vertical and horizontal synchronization signals required by the monitor.

The bit clock is disabled by vertical and horizontal blanking, and its rate can be set by the microcode to either 50 or 100 ns. by the function SETMODE (F2=11B, specific to the display horizontal task DHT). This function examines the two high order bits of the processor bus. If bit 0=1, the bit clock rate is set to 100ns period (at the start of the next scan line), and a 1 is merged into NEXT[9]. SETMODE also latches bit 1 of the processor bus and uses the value to control the polarity of the video output. A third function, EVENFIELD (F2=10B, specific to DHT and to the display vertical task DVT), merges a 1 into NEXT[9] if the display is in the even field.

The display control hardware also generates wakeup requests to the microprocessor tasking hardware. The vertical task DVT is awakened once per field, at the beginning of vertical retrace. The display horizontal task is awakened once at the beginning of each field, and thereafter whenever the display word task blocks. DHT can block itself, in which case neither it nor the word task can be awakened until the start of the next field. The wakeup request for the display word task (DWT) is controlled by the state of the 16 word buffer. If DWT has not executed a BLOCK, if DHT is not blocked, and if the buffer is not full, DWT wakeups are generated. The hardware sets the buffer empty and clears the DWT block flip-flop at the beginning of horizontal retrace for every scan line.

4.3 Display Controller Microcode

The display controller microcode is divided into three tasks. The highest priority task is DVT, the display vertical task, the next is DHT, the horizontal task, and the third is DWT, the display word task. The display controller uses 6 registers in R:

CBA:	Holds the address of the currently active DCB+1.
AECL:	Holds the address of the end of the currently active scan line's bit map in main memory.
SLC:	Holds the number of scan lines remaining in the currently active DCB.
HTAB:	Holds the number of tab words remaining on the current scan line.
DWA:	Holds the address of the bit map doubleword currently being fetched for transmission to the hardware buffer.
MTEMP:	Is a temporary cell.

The vertical task initializes the controller by setting SLC to 0 and CBA to DASTART+1. It also merges the contents of DASTART+1 into NWW, which will cause an interrupt if the specified channel is active. DVT also sets up information required for the cursor (see below), TASKs and becomes inactive until the next field.

DHT starts by initiating a fetch to the word addressed by CBA. It checks SLC, and if it is zero, the controller is finished with the current DCB, and the link word of the DCB is fetched. If this word is non-zero, it replaces CBA and processing of a new DCB is begun. If the link word is zero, DHT blocks until the start of the next field.

If the check of SLC indicates that more scan lines remain in the current DCB, SLC is decremented by one and the fetch of (CBA) is used to obtain the second word of the DCB, rather than the link word. The contents of this word are used to set the display mode and polarity, and the tab count is extracted and put into HTAB. NWRDS is extracted, and used to increment DWA and AECL by the appropriate amount, depending on the mode and field. All the registers required by DWT have now been set up, and DHT TASKs and becomes inactive until DWT blocks.

If a new DCB is required, DHT fetches all four words of the new DCB, and initializes all the registers. During all scan lines of a DCB except the first, DHT only accesses the first doubleword of the block.

DWT has the sole task of transferring words from memory to the hardware. When it first awakens during horizontal retrace, it checks HTAB. If it is non-zero, it enters a loop which outputs HTAB 0's to the display. When HTAB is zero, a second loop is entered which fetches a doubleword from the location specified by DWA. DWA is compared with AECL, and if they are equal, DWT blocks until the next scan line. DWA is incremented by 2, in preparation for the fetch of the next doubleword. If DWA≠AECL, DWT continues to supply words to the buffer whenever it becomes non-full.

4.4 Cursor

Because of the difficulty of inserting a cursor at the appropriate place in the display bit map at reasonable speed, a hardware cursor is included in the Alto. The cursor consists of an arbitrary 16x16 bit patch, which is merged with the video at the appropriate time. The bit map for the cursor is contained in 16 words starting at location CURMAP(431B) in page one, and the x,y coordinates of the cursor are specified by location CURLOC (426B) and CURLOC+1 (427B) in page 1. The coordinate origin for the cursor is the upper left hand corner of the screen. The cursor presentation is unaffected by changes in display resolution. Its polarity is that of the current DCB, or the last DCB processed if it is located on an area of the screen not defined by a DCB. The cursor may be removed from view in a number of ways. The most efficient in terms of processing time is to set the x coordinate to -1.

The cursor hardware consists of a 16-bit shift register which holds the information to be displayed on the current scan line, and a counter which is incremented by the bit clock, and determines the x coordinate at which the shift register begins shifting.

The hardware is loaded during horizontal retrace by the cursor task microcode, which simply copies the x coordinate and bit map segment from the R memory into the hardware.

The values of x and the bit map are set up in R by a section of the memory refresh task, whose wakeup and priority are arranged so that it runs during every scan line after DWT has done all necessary output and DHT has set up the information required by DWT for the next scan line. MRT checks the current y position of the display, and if it is in the range in which the cursor should be displayed, fetches the appropriate bit map segment from CURMAP. When the cursor y position is exceeded by the display, a flag is set in MRT to disable further processing. The x and y coordinates of the cursor are fetched from CURLOC and CURLOC+1 at the beginning of each display field by a section of the display vertical task microcode.

Cursor processing is distributed as it is to minimize the amount of processing which must be done during the monitor's horizontal retrace time. This time is approximately 6 microseconds, and it must include the worst case latency imposed by tasks at lower priority than the display, plus the worst case disk word processing time (the disk word task is at higher priority than the display), plus the time necessary for DWT to partially fill the display buffer, plus cursor processing time.

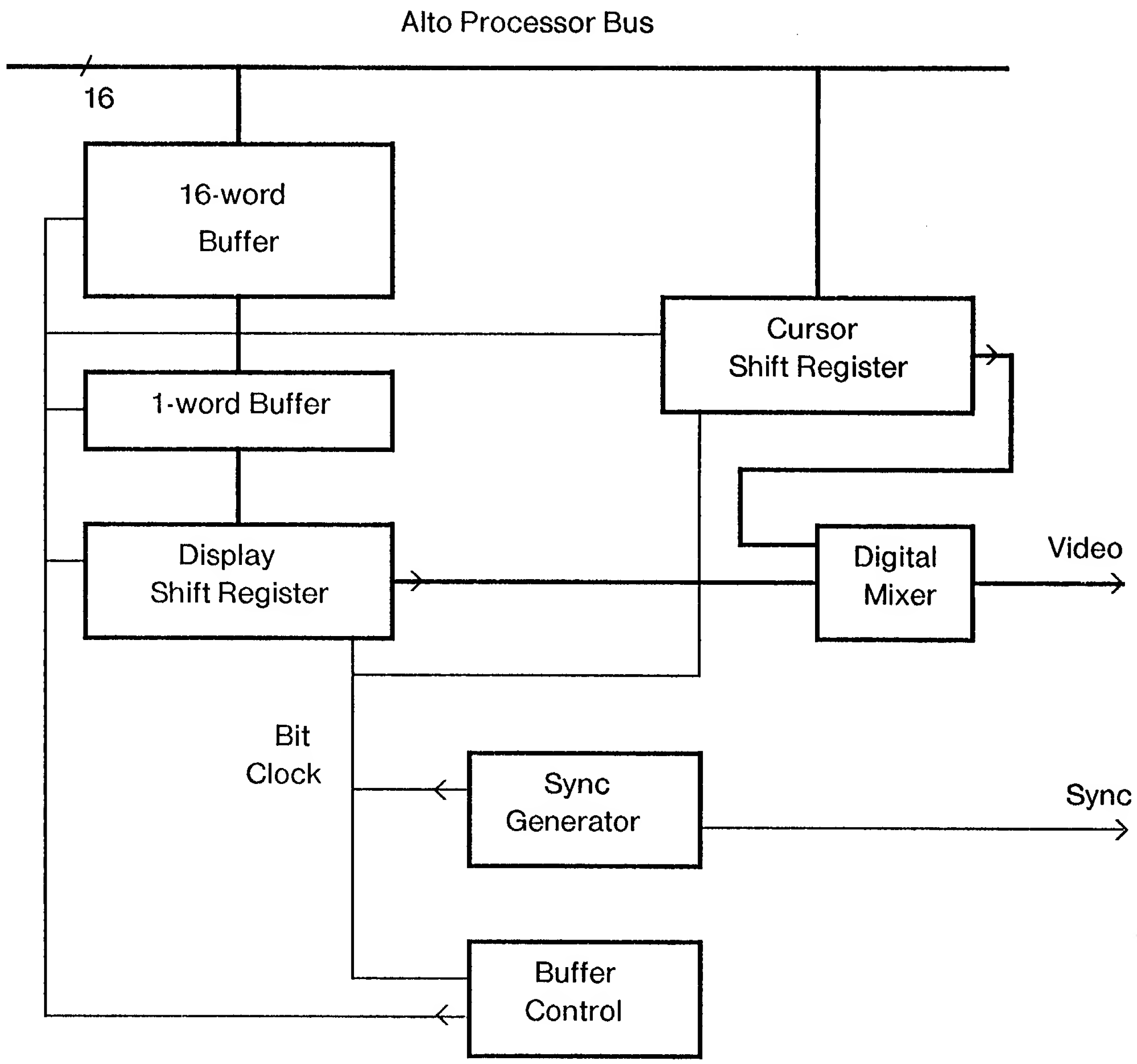
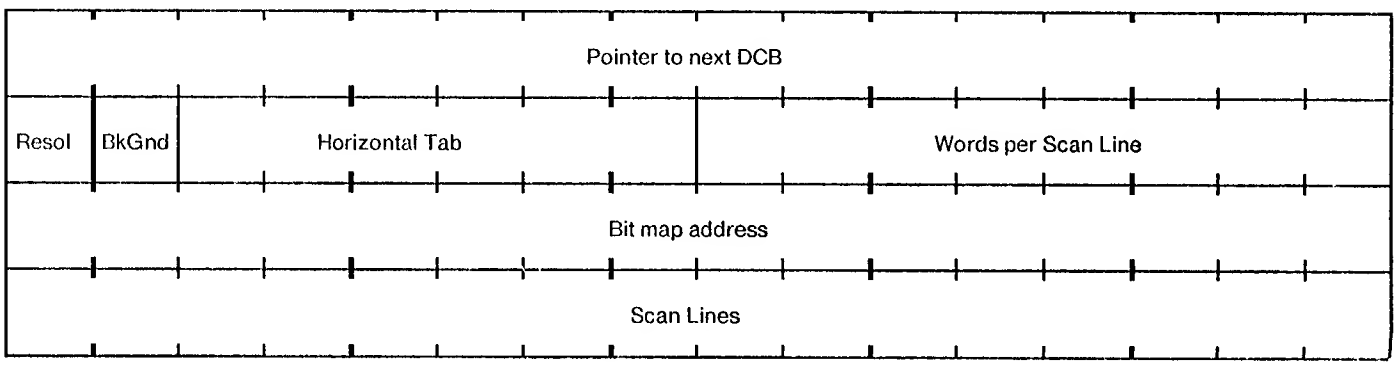


Figure 5 -- Display Control


Pointer to next DCB
Resol
BkGnd
Horizontal Tab
Words per Scan Line
Bit map address
Scan Lines